

Binary Search Trees

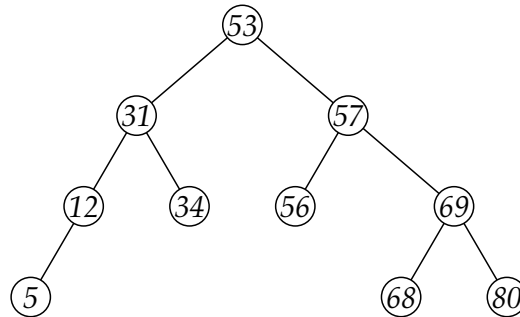
A *binary search tree* is used to store ordered data to allow efficient queries and updates.

14.1 Binary Search Trees

A *binary search tree* is a binary tree with values at the nodes such that

left descendants are smaller (or equal), right descendants are bigger (or equal).

This assumes the data comes from a domain in which there is a *total order*: you can compare every pair of elements (and there is no inconsistency such as $a < b < c < a$). In general, we could have a large object at each node, but the objects are sorted with respect to a *key*.



An *inorder traversal* is when a node is visited after its left descendants and before its right descendants. The following recursive method is started by the call `inorder(root)`.

```

void inorder(Node *v) {
    inorder ( v->left );
    visit v;
    inorder ( v->right );
}
  
```

An inorder traversal of a binary search tree prints out the data in order.

14.2 Insertion in BST

To find an element, you compare it with the root. If larger, go right; if smaller, go left. And repeat. The following method returns NULL if not found:

```
Node *find(key x) {
    Node *t=root;
    while( t!=NULL && x!=t->key )
        t = ( x<t->key ? t->left : t->right );
    return t;
}
```

Insertion is a similar process to searching, except you need a bit of look ahead. Here is a strange-looking *recursive* version:

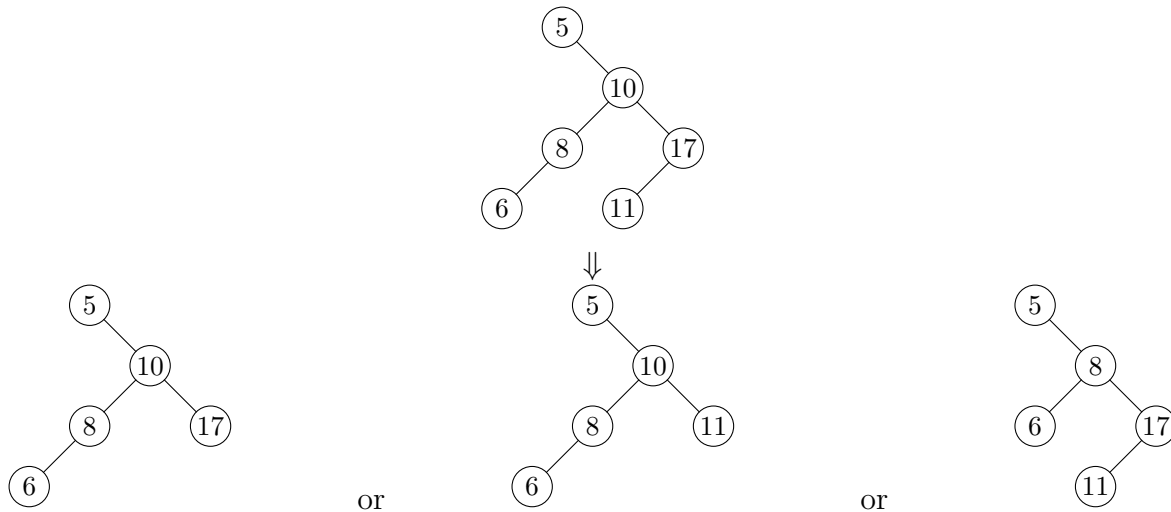
```
Node *insert(ItemType &elem, Node *t) {
    if( t==NULL )
        return new Node( elem );
    else {
        if( elem.key<t->key )
            t->left = insert(elem,t->left);
        else if( elem.key>t->key )
            t->right = insert(elem,t->right);
        return t;
    }
}
```

14.3 Removal from BST

To remove a value, one first finds the node that is to be removed. The algorithm for removing a node is divided into three cases:

- *Node is a leaf.* Then just delete.
- *Node has only one child.* Then delete the node and do “**adoption by grand-parent**” (get old parent to point to old child).
- *Node x has two children.* Then find the node y with the next-lowest value: go left, and then go repeatedly right (why does this work?). This node y cannot have two children (in fact it does not have a right child). So swop the values of x and y , and delete the node y using one of the previous cases.

The following picture shows a binary search tree and what happens if 11, 17, or 10 (assuming replace with next-lowest) is removed.



All modification operations take time proportional to depth. In best case, the depth is $O(\log n)$ (why?). But, the tree can become “lop-sided”—and so in worst case these operations are $O(n)$.

14.4 Finding the k 'th Largest Element in a Collection

Using a binary search tree, one can offer the service of finding the k 'th largest element in the collection. The idea is to keep track at each node of its size (how many nodes counting it and its descendants). This tells one where to go.

For example, if we want the 4th smallest element, and the size of the left child of the root is 2, then the value is the minimum value in the right subtree. (Why?) (This should remind you of binary search in an array.)

14.5 Sample Code: BinarySearchTree

```
// BSTNode.h
// provides node suitable for BST
// wdg 2009
#ifdef BSTNODE_H
#define BSTNODE_H
#include <cstdlib>

typedef char ItemType;
```

```

struct BSTNode
{
    BSTNode (ItemType e, BSTNode *p) :
        element(e), parent(p), left(NULL), right(NULL)
    {}
    ItemType element;
    BSTNode *left, *right, *parent;
};
#endif



---



// implementation of BST
// wdg 2009
#ifdef BINARY_SEARCH_TREE_H
#define BINARY_SEARCH_TREE_H
#include "BSTNode.h"

class BinarySearchTree
{
public:
    BinarySearchTree( );
    virtual void insertItem(ItemType it); // to allow extension
    bool deleteItem(ItemType it);
    bool findItem(ItemType it) const;
    void dumpItemsInOrder( ) const;

protected:          // to enable extension to RedBlack trees
    BSTNode *root;
    int count;

    bool insertItem(BSTNode *newNode);
    BSTNode *search( ItemType it ) const;
    void inOrder(BSTNode *v) const;
};
#endif



---



// implementation of BST
// does not allow duplicate items
// wdg 2008

#include <iostream>

```

```

#include "BinarySearchTree.h"
using namespace std;

// constructor creates an empty tree

    BinarySearchTree::BinarySearchTree( ) : root(NULL) , count(0)
    {
    }

//=====
// GENERAL BINARY SEARCH TREE METHODS
//=====
// method search starts at root looking for that item
// returns either node where found, or failing that,
// position where should be attached
// assumes root!=null

BSTNode *BinarySearchTree::search( ItemType it ) const
{
    BSTNode *v=root;
    bool absent=false;
    while( !absent && it!=v->element ) {
        if( it<v->element && v->left!=NULL )
            v = v->left;
        else if( it>v->element && v->right!=NULL )
            v = v->right;
        else
            absent = true;
    }
    return v;
}

//-----
// inserts item

void BinarySearchTree::insertItem(ItemType it)
{
    insertItem( new BSTNode(it,NULL) );
}

bool BinarySearchTree::insertItem(BSTNode *newNode)

```

```

{
    ItemType it = newNode->element;
    if(count==0) {
        count=1;
    root=newNode;
    }
    else {
        BSTNode *v = search(it);
        if( it < v->element ) {
            count++;
            v->left=newNode;
            newNode->parent = v;
        }
        else if( it > v->element ) {
            count++;
            v->right = newNode;
            newNode->parent = v;
        }
        else {
            cout << it << " duplicate" << endl;
            return false;
        }
    }
    return true;
}
//-----
// find item

bool BinarySearchTree::findItem(ItemType it) const
{
    if(count==0)
        return false;
    ItemType item = search(it)->element;
    if( item == it )
        return true;
    else
        return false;
}
//-----

```

```

// delete item

bool BinarySearchTree::deleteItem(ItemType it)
{
    BSTNode *toBeDeleted=NULL;
    if( root==NULL)
        return false;
    else if( it == root->element && ( root->left==NULL || root->right== NULL ) ) {
        // root deleted
        toBeDeleted = root;
        if( root->left!=NULL )
            root = root->left;
        else
            root = root->right;
        count--;
        delete toBeDeleted;
        return true;
    }
    else { // root not deleted
        BSTNode *parent = NULL;
        BSTNode *curr = root;
        while ( curr!=NULL && curr->element!=it) {
            parent = curr;
            if( it < curr->element )
                curr = curr->left;
            else
                curr = curr->right;
        }
        if( curr==NULL )
            return false;
        else {
            if( curr->left!=NULL && curr->right!=NULL ) {
                BSTNode *hold = curr;
                parent = curr;
                curr = curr->right;
                while ( curr->left!=NULL ) {
                    parent = curr;
                    curr = curr->left;
                }
            }
        }
    }
}

```

```

        hold->element = curr->element;
    }
    toBeDeleted = curr;
    if( curr->left!=NULL )
        curr = curr->left;
    else
        curr = curr->right;
    if( toBeDeleted == parent->left )
        parent->left = curr;
    else
        parent->right = curr;
    count--;
    delete toBeDeleted;
    return true;
}
} // end-if root not deleted
}

//=====
// OUTPUT METHODS
//=====

void BinarySearchTree::dumpItemsInOrder( ) const
{
    if( count==0 )
        cout << "Empty" << endl;
    else {
        inOrder(root);
        cout << endl;
    }
}

void BinarySearchTree::inOrder(BSTNode *v) const
{
    if( v==NULL )
        return;
    inOrder(v->left);
    cout << v->element;
    inOrder(v->right);
}

```